# FTPM Context

*How does Fast Time Plot plug in?*

Mon, Apr 16, 2001

The Acnet Fast Time Plot protocol is commonly referred to as `FTPMAN` by its Acnet task name. In the IRM front-ends, this protocol is supported by a local application called `FTPM`, which resides in the program file `LOOPFTPM`. Like all program files, it is a function that is separately compiled and downloaded into the front-end. When its associated software enable bit is set, it is brought into execution to become an extension of the underlying system software as long as the enable bit remains set. If the enable bit should be cleared, it will free its resources and no longer be part of the system code operation. This note describes in more detail how this connection to the system code works. The same scheme applies for any network protocol whose support is handled by a local application.

The front-end system code operates at a typical rate of 15 Hz, in synchronism with the Fermilab accelerator operation. A predictable sequence of tasks are executed every cycle to serve as a framework of that front-end's control system support. The first main task performed in each cycle is the `Update` task, which updates the data pool by acquiring fresh data from its hardware interfaces and fulfills all active data requests, often using its freshly-acquired device readings in the data pool. replies to data requests are therefore delivered periodically in synchronism with the accelerator at sub-multiples of 15 Hz or on cycles when a selected clock event has occurred since the previous cycle. Except for clock event-based requests, there is also a prompt reply that is fulfilled and delivered as soon as a new request has been accepted and initialized. For one-shot requests, this prompt reply is the only reply delivered, after which resources needed by that one-shot request are freed.

The context for any local application function is that it is invoked by passing it two arguments. The first is a small index number that provides the reason for the call. The second is a pointer to the local application table entry that houses the parameter structure used by this application instance. (One might have multiple instances of a closed loop algorithm, for example, each of which is passed a set of parameters that specify its related signals. For local applications that provide support for network protocols, multiple instances would not apply.) The parameter structure includes a pointer that is established the first time the local application is called, followed by the enable bit number followed by up to nine additional 16-bits words whose meaning is known only to the application code and to the user who configures the application instance.

The possible calls to a local application are limited. The first `Initialization` call it gets is made by the `Update` task when it has detected the enable bit in use by a particular `LATBL` entry becoming set. In response to this call, the LA should allocate a context block that can serve as its own context memory as long as it is active. As long as the enable bit remains set, a `Cycle` call will be made at the `Update` task every cycle during Data Access Table processing when it encounters the instruction that means, "invoke all the enabled local applications." If the enable bit should ever be cleared, the `Update` task will invoke it one last time with a `Termination` call, which should cause it to free its allocated resources. For LAs that support network protocols, another

invocation type can occur, the `Net` call. If the protocol is an Acnet header-based protocol, this call is made from the `AcReq` task, which includes built-in support for the Acnet protocols `RETDAT` and `SETDAT`. If the LA supports a UDP port-based protocol, the call is made from the task that manages such communications, which is `SNAP` in the pSOS-based systems or `tSRd` in the vxWorks-based systems.

Limiting the discussion to Acnet protocols, how does the `AcReq` task know to invoke the LA? The `AcReq` task waits on a message queue called `ACRQ`. So the trick is to make sure that the appropriate reference message is written to this message queue when an Acnet message arrives that uses the corresponding Acnet protocol. This will awaken `AcReq`, and since the protocol is not `RETDAT` nor `SETDAT`, it will look for the appropriate LA to invoke to handle the message.

But how does a reference message get written into the message queue upon which `AcReq` waits? The Net Connect table, or `NETCT`, is used for this purpose. When an Acnet protocol is opened via `NetCnct` or the higher-level `NetOpen`, an entry is placed into the `NETCT` that relates the Acnet protocol task name to the message queue id. The `ANet` task is the Acnet protocol dispatcher, seeing to it that any Acnet protocol message is directed to the proper task. `ANet` waits on the Acnet UDP port number 6801. The system network socket support delivers a reference message about a received Acnet datagram to `ANet`. And `ANet` examines the datagram in sequence, as multiple Acnet header-based messages can be concatenated within a single Acnet datagram. For each message it determines the destination. If the message type is a `USM` or `REQ`, the task name field in the Acnet header announces the intended receiver, leading to the matching `NETCT` entry. If the message type is a `RPY`, then the source task id field identifies the `NETCT` entry directly. In either case, a reference message is built and written to the message queue whose queue id is found in that `NETCT` entry.

Note that for any Acnet protocol, the queue id will be the same one upon which `AcReq` waits. So how can an LA pass to `NetCnct` or `NetOpen` the message queue id in use by `AcReq`? It presumes to know its name, `ACRQ`, so it gets the message queue id from the operating system. In the system source code, this is a function like `QAttach`.

Now we have a scheme that provides a means of passing a network message to a LA whose job it is to support an arbitrary Acnet protocol. But this is not enough. We also need to get "plugged into" the system in a way that the LA can know when to fulfill a data request and queue it to the network. To this end, a description of how request fulfillment is in order.

All active data requests are fulfilled by the `Update` task early in the cycle as soon as the data pool has been refreshed. To this end, all active data requests are organized into a linked list of allocated request blocks, a different type used for each protocol type. When any data request is initialized, a request block is allocated and its fields initialized. Then the block is added to the linked list. It is not merely added to the end, or the beginning, of the linked list, but rather it is added to a place in the linked list adjacent to another request block associated with a request made by the same node. (If there are no other active requests from that node, the request block is placed at the head of the linked list.) The reason for maintaining this order of the linked list is to facilitate

the combining of multiple messages into common datagrams. The `Update` task, in fulfilling all active data requests that are due on a given cycle, processes the linked list in order. This ordering scheme means that if more than one reply message is to be built to return to the same requesting node on the same cycle, then the reply messages will be queued to the network in sequence. When the network queue is flushed, the generic code that builds datagrams for passing to the socket transmit code combines consecutive messages that target the same node into a common datagram as best it can.

So we need a means for the `Update` task to learn what routine to call to fulfill a request, based upon the protocol. Since each request block for different protocols can be of a different type, the key is the block type number. Each request block uses a common 8-byte header of the following form:

```
short mBlkSize;
long linkNext;
short mBlkType;
```

The `mBlkType` field is the block type number, using values in the range `1—7F`. The `Update` task needs to use the block type number to decide what routine to call, including an LA if it is not a protocol for which built-in support is provided. The means of accomplishing this is another table, the `PROTO` table. An LA that wants to support an Acnet protocol needs to announce its intention to do so by invoking `OpenPro` to establish an entry in the `PROTO` table. This entry includes a block type number, the Acnet task name, a pointer to an "`Update`" routine provided by the LA, and a pointer to the `LATBL` entry, the same pointer that is passed to the LA whenever it is invoked. The "`Update`" routine is a separate function that is passed two arguments. The first is a call type number and the second is a pointer to the request block.